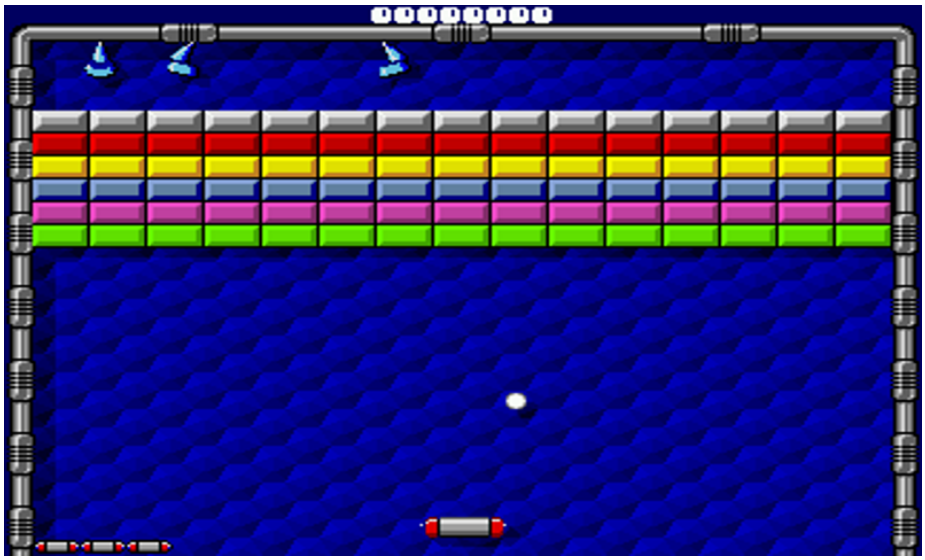
Arkanoid

Arkanoid ou Breakout é um jogo lançado pela Taito em 1986 em que o jogador controla uma nave chamada *Vaus* que se move horizontalmente rebatendo uma bola para que atinja obstáculos na parte de cima da tela. O objetivo do jogo é remover todos esses obstáculos sem que a bola caia na parte de baixo da tela.



Na aula de hoje iremos implementar um clone desse jogo para reforçar os conceitos aprendidos até agora da linguagem Haskell.

Esse jogo foi baseado na implementação encontrada em https://github.com/CrackYSK/Arkanoid.

Instalação das bibliotecas no Linux

Utilizando o gerenciador de pacotes da sua distribuição instale o pacote equivalente ao `freeglut3-dev`

```
sudo apt-get install freeglut3-dev
```

Instalação das bibliotecas no MS Windows

O jogo utilizará a biblioteca GLUT. Para instalá-la:

1) Baixe o Glut de http://www.transmissionzero.co.uk/software/freeglut-devel/

2) Copie o arquivo `freeglut-MinGW-<versao>.mp.zip\freeglut\bin\x64\freeglut.dll` para o diretório `C:\Windows\System32`

3) Renomeie o arquivo copiado para glut32.dll

(Fonte: Stack Overflow)

Jogos no Haskell

Para construir jogos simples no Haskell itemos utilizar a biblioteca Gloss. Criem um projeto chamado **Arkanoid** com o Stack:

```
> stack new Arkanoid simple
```

Em seguida, edite o arquivo *Arkanoid.cabal* alterando as linhas *build-depends* e *other-modules* para:

```
build-depends:       base >= 4.7 && < 5, gloss
other-modules:       Window
```

Na pasta do projeto digite (no windows, a primeira linha deve ser criar um arquivo Window.hs vazio na pasta src):

```
> touch src/Window.hs
> stack setup
> stack build
```

Criando a tela do jogo

No arquivo *Window.hs* digite:

```
module Window where

import Graphics.Gloss
```

A primeira linha indica que esse é o módulo Window e poderá ser importado em qualquer outro código-fonte do projeto através do comando `import`. A segunda linha importa a biblioteca *Gloss* para usarmos suas declarações e funções.

Ainda no arquivo *Window.hs* vamos escrever as definições de nossa janela. Para criar uma janela no Gloss primeiro criamos um valor do tipo `Display` com o construtor `InWindow`:

```
InWindow :: String -> (Int, Int) -> (Int, Int) -> Display
```

O primeiro parâmetro é o título da janela, o segundo é a informação de largura e altura, e o último parâmetro é a posição da janela:

```
-- Configuração da tela
```

```haskell
-- | Largura da janela.
width :: Int
width = 800

-- | Altura da janela.
height :: Int
height = 600

-- | Posição da janela.
offset :: Int
offset = 100

-- | Cor de fundo.
background :: Color
background = white

-- | Janela do jogo.
window :: Display
window = InWindow "Arkanoid" (width, height) (offset, offset)
```

No arquivo *Main.hs* escreva:

```haskell
-- | Arkanoid game implemented in Haskell.
module Main where

import Graphics.Gloss
import Window

-- | Window creation.
main :: IO ()
main = display window background (circleSolid 10)
```

A função `display` é do tipo:

```haskell
display :: Display -> Color -> Picture -> IO ()
```

Ou seja, necessita de uma janela, uma cor de fundo e uma imagem para desenhar dentro da janela. O tipo de retorno `IO ()` indica que ela faz alguma interação de entrada e saída com o ambiente, nesse caso, dos *display* gráfico. A função `circleSolid` constrói um círculo preenchido de raio `n` no centro da janela.

Exercício 01: altere os valores de `width`, `height`, `offset`, `background` e verifique os resultados.

Para alterar a cor de algum objeto, utilizamos a função `color` que recebe uma cor, um objeto de imagem e retorna esse objeto alterado:

```haskell
color :: Color -> Picture -> Picture
```

Similarmente, podemos aplicar a função `translate` que altera a posição do objeto:

```
trasnlate :: Float -> Float -> Picture -> Picture
```

Note que a coordenada central da janela é $(0, 0)$, portanto a faixa de valores aceitáveis é $(-width/2, width/2)$ para a horizontal e $(-height/2, height/2)$ na vertical. Altere a função `main` para:

```
main :: IO ()
main = display window background (translate (-10) 10 $ color red $ circleSolid 10)
```

O operador `$` funciona como um *pipe* de operações, e é equivalente a:

```
translate (-10) 10 (color red (circleSolid 10))
```

Exercício 02: altere os parâmetros de `translate` e `color` e verifique o resultado.

Vocês devem ter notado que os eixos x e y são iguais aos eixos cartesianos, valores negativos são pontos a esquerda e para baixo no eixo.

Como frequentemente teremos que referenciar os extremos da janela, criem as seguintes declarações no arquivo *Window.hs*:

```
-- | Para uso nas funções de colisão.
-- Esse valor é o extremo da tela
halfWidth :: Float
halfWidth = fromIntegral width / 2

halfHeight :: Float
halfHeight = fromIntegral height / 2
```

Para construir as bordas, podemos desenhar retângulos na parte de cima, esquerda e direita da janela:

```
-- | Cor da borda.
wallColor :: Color
wallColor = green

-- | Borda de cima.
topWall :: Picture
topWall = translate 0 halfHeight
        $ color wallColor
        $ rectangleSolid (fromIntegral width) 10
```

A função `rectangleSolid` recebe dois `Float` como argumentos, portanto devemos converter `width` para o tipo correto utilizando a função `fromIntegral`. Para testar, altere a função `main` para:

```haskell
-- | Window creation.
main :: IO ()
main = display window background topWall
```

Exercício 03: Desenhe as bordas laterais `leftWall` e `rightWall`.

Para juntar as três bordas e desenhar todas juntas na janela, basta utilizar a função `pictures`:

```haskell
pictures :: [Picture] -> Picture
```

No arquivo *Window.hs* faça:

```haskell
-- | Imagem das bordas.
walls :: Picture
walls = pictures [leftWall, rightWall, topWall]
```

E altere a função `main` para:

```haskell
-- | Window creation.
main :: IO ()
main = display window background walls
```

Para escrever um texto na tela, utilizamos a função `text` juntamente com a função `scale` para ajustar o tamanho da imagem:

```haskell
text  :: String -> Picture
scale :: Float -> Float -> Picture -> Picture
```

Exercício 04: Crie uma função `renderTxt` no arquivo *Window.hs* que recebe uma cor (tipo `Color`) e uma `String` e retorna um texto escalado em $0.30.3$ e na posição $(-150, 150)$. Altere a `main` para testar a função.

Acrescente as seguintes declarações no arquivo *Window.hs*:

```haskell
-- | Mensagem atual a ser mostrada.
curMsg :: Int -> Bool -> Picture
curMsg  0   paused = pauseMsg paused
curMsg (-1) paused = lostMsg
curMsg  _   paused = winMsg

winMsg    = renderTxt green "You won! (r = new game)"
lostMsg   = renderTxt red   "Git gud! (r = new game)"
```

```
pauseMsg True  = renderTxt blue "Press p to play!"
pauseMsg False = renderTxt blue ""
```

Criando a bola

Crie o arquivo *Ball.hs* no diretório *src* e, no arquivo *Arkanoid.cabal*, altere a linha *other-modules* para:

```
other-modules:       Window, Ball
```

No arquivo *Ball.hs* inclua:

```
module Ball where

import Graphics.Gloss

-- | Coordenada do centro da bola.
type Position = (Float, Float)

-- Propriedades da bola

-- | Raio da bola.
ballSize :: Float
ballSize = 5

-- | Cor da bola.
ballColor :: Color
ballColor = red
```

Exercício 05: Crie uma função `ball` que recebe um tipo `Position` e retorna o desenho de uma bola de raio `ballSize` e cor `ballColor`:

```
-- | Cria a imagem da bola no estado atual do jogo.
ball :: Position -> Picture
ball (x, y) = undefined
```

No arquivo *Main.hs* importe o módulo *Ball* e na função `main` escreva:

```
main = display window background (pictures [walls, ball (0,0)])
```

Exercício 06: Agora crie uma função `moveBall` que recebe uma quantidade de segundos, a posição e a velocidade da bola e retorna a nova posição dela seguindo a equação $s = s_0 + v \cdot t$:

```
moveBall :: Float -> Position -> Position -> Position
moveBall segundos (x, y) (vx, vy) = (x', y')
  where
    x' = undefined
    y' = undefined
```

Isso é tudo para esse arquivo!

Blocos

Acrescente um arquivo *Blocks.hs* ao projeto (lembre-se de alterar o arquivo cabal) contendo:

```
module Blocks where

import Graphics.Gloss
import Ball


-- Propriedades dos blocos
-- | Blocos por fileira
blocksPerRow :: Int
blocksPerRow = 15

-- | Tamanho dos blocos.
blockSize :: (Float, Float)
blockSize = (20, 10)

bHalfWidth :: Float
bHalfWidth  = (1 + fst blockSize) / 2

bHalfHeight :: Float
bHalfHeight = (1 + snd blockSize) / 2

-- | Informação dos blocos.
data BlockInfo = Block
  { blockPos :: Position -- ^ (x, y) coordenada do bloco.
  , blockCol :: Color    -- ^ cor do bloco.
  }
```

```haskell
-- | Lista dos blocos atuais.
type Blocks = [BlockInfo]
```

Para essa parte do código criaremos três funções: `hasBlocks` que verifica se existe ainda algum bloco na lista de blocos, `genBlock` que gera as informações de cada um dos 60 blocos do jogo e `drawBlocks` que desenha uma lista de blocos.

Exercício 07: Crie a função `hasBlocks` que recebe uma lista de blocos e retorna um `Bool` indicando se ela não está vazia:

```haskell
-- | Verifica se ainda existem blocos a serem destruídos.
hasBlocks :: Blocks -> Bool
hasBlocks blocks = undefined
```

Exercício 08: Crie a função `drawBlocks` que desenha cada bloco (utilizando `translate`, `color`, `rectangleSolid`) de uma lista de blocos e retorna uma imagem única com a função `pictures`:

```haskell
-- | Desenha os blocks.
drawBlocks :: Blocks -> Picture
drawBlocks bs = pictures $ undefined
  where
    drawBlock (Block (x, y) col) = undefined
    block                        = undefined
    (w, h)                       = blockSize
```

Exercício 09: Crie a função `genBlock` que mapeia o *id* de um bloco e gera sua informação de posição e cor (todos terão a mesma cor nessa versão). Imaginando um quadriculado de 15×4 blocos, e que começamos a desenhar na coordenada (-250, 100), temos que a posição (bx, by) do bloco é $bx = -250 + (nmod blocksPerRow) * 35$ e $by = 100 - (ndiv blocksPerRow) * 40$, os números $35, 40$ representam a largura e altura dos blocos acrescidos de um pequeno espaço entre eles:

– | Gera um dos 60 blocos iniciando da coordenada (-250, 100) genBlock :: Int -> BlockInfo genBlock n = Block { blockPos = pos, blockCol = orange } where pos = (fromIntegral bx, fromIntegral by) bx = undefined by = undefined (y, x) = n `divMod` blocksPerRow “‘

Para testar faça na função `main`:

```
main = display window background (pictures [walls, ball (0,-100),
                                            drawBlocks (map genBlock [0..59])])
```

Desenhando o jogador

A representação gráfica do jogador é simplesmente um retângulo controlado pelo usuário com o objetivo de rebater a bola. Adicione um arquivo *Player.hs* ao projeto e adicione o seguinte código:

```
module Player where

import Graphics.Gloss
import Window

-- Informações do jogador

-- | Cor do jogador.
playerColor :: Color
playerColor = blue

-- | Tamanho do jogador.
playerWidth :: Float
playerWidth = 50

halfPlayerWidth :: Float
halfPlayerWidth = playerWidth / 2

playerHeight :: Float
playerHeight = 10

-- | Posição do jogador no eixo y
playerY :: Float
playerY = -250
```

Exercício 10: Crie a função `mkPlayer` que recebe a coordenada x do jogador e retorna a figura correspondente. Utilize `rectangleSolid` para desenhar a forma:

```
-- | Imagem do jogador.
mkPlayer :: Float -> Picture
mkPlayer x = undefined
```

Exercício 11: Crie a função `movePlayer` que recebe a quantidade de segundos, a posição atual e a velocidade (horizontais) e retorna a nova posição horizontal seguindo a equação $x = x_0 + v \cdot t$:

```
movePlayer :: Float -> Float -> Float -> Float
movePlayer seconds x v = undefined
```

Exercício 12: Um problema com essa função é que se o jogador ultrapassar as bordas, ela permitirá que ele saia para fora da janela. Vamos criar três funções que detectam se o jogador atingiu a borda da esquerda e da direita, note que as bordas se localizam em `-halfWidth` e `halfWidth` e as pontas do jogador em `x - halfPlayerWidth` e `x + halfPlayerWidth`:

```
-- | Verifica se o jogador atingiu a parede da esquerda.
leftWallCollision :: Float -> Bool
leftWallCollision x | x - halfPlayerWidth <= -halfWidth + 5  = True
                    | otherwise                              = False

-- | Verifica se o jogador atingiu a parede da direita.
rightWallCollision :: Float -> Bool
rightWallCollision x | undefined
                     | otherwise                              = False
```

Exercício 13: Crie a função `paddleWallCollision` que determina se o jogador colidiou com a borda da esquerda OU a borda da direita:

```
-- | Verifica se o jogador atingiu a parede.
paddleWallCollision :: Float -> Bool
paddleWallCollision x = undefined
```

Exercício 14: Altere a função `movePlayer` de tal forma que se ele não colidiu com as bordas, retorne a posição atualizada, porém se colidir com a direita e a velocidade for positiva, retorne $x - 1$ e se colidir com a esquerda e a velocidade for negativa, retorne $x + 1$.

```
-- | Atualiza posição do jogador.
-- | Atualiza posição do jogador.
movePlayer :: Float -> Float -> Float -> Float
movePlayer seconds x v | condicao1                    = x - 1
                       | condicao2                    = x + 1
                       | otherwise                    = undefined
  where deltaX = undefined
```

Detectando Colisões

Adicione um arquivo *Collision.hs* ao projeto e inclua o seguinte código (solução do exercício para casa):

```
module Collision where

import Ball
import Blocks
import Window
import Player

-- | Função para if-then-else.
funIf :: Bool -> a -> a -> a
funIf b x y = if b then x else y

-- | Multiplica elementos de duas tuplas.
mulTuple :: Num a => (a, a) -> (a, a) -> (a, a)
mulTuple (x1,x2) (y1,y2) = (x1*y1, x2*y2)

-- | Verifica se tem interseção entre duas faixas de valores.
overlap :: Ord a => (a, a) -> (a, a) -> Bool
overlap (xmin, xmax) (ymin, ymax) = xmin <= ymax && ymin <= xmax

-- | Cria uma faixa de valores centrado em x e com raio r.
range :: Num a => a -> a -> (a, a)
range x r = (x - r, x + r)

-- | Retorna se a bola colidiu com uma das bordas.
topCollision :: Position -> Bool
```

```
topCollision (x, y)   = y + 2 * ballSize >   halfHeight

leftCollision :: Position -> Bool
leftCollision (x, y)  = x - 2 * ballSize <= -halfWidth

rightCollision :: Position -> Bool
rightCollision (x, y) = x + 2 * ballSize >=  halfWidth

-- | Retorna se a bola colidiu com o jogador.
paddleCollision :: Float -> Position -> Bool
paddleCollision playerX (x, y) = yCollision && xCollision
  where
    yCollision = y - ballSize <= playerY  && y - 1 >= playerY
    xCollision = x >= playerX - 25 && x <= playerX + 25

-- | Verifica se atinge a borda de algum bloco
inCorner :: (Num a, Ord a) => a -> (a, a) -> (a, a) -> Bool
inCorner x (xmin, xmax) (rmin, rmax) = (xmin > x + rmin && xmin < x + rmax)
                                    || (xmax < x - rmin && xmax > x - rmax)
```

Agora precisamos acrescentar apenas quatro funções: `blockCollision`, `removeBlocks` que altera a velocidade da bola ao atingir um bloco e remove os blocos atingidos, `paddleBounce`, `wallBounce` que altera a velocidade da bola ao atingir o jogador ou a borda da tela.

Exercício 15: Crie a função `wallBounce` que recebe a posição da bola e sua velocidade (vx, vy) e retorna (-vx, vy) se ele colidiu APENAS com a bordas laterais, (vx, -vy) se ele colidiu com o topo, e (vx, vy) caso contrário:

```
-- | Detecta colisão da bola com as bordas e atualiza velocidade.
wallBounce :: Position -> Position -> Position
wallBounce pos (vx, vy) | condicao1    = (-vx,  vy)
                        | condicao2    = ( vx, -vy)
                        | otherwise    = ( vx,  vy)
```

Exercício 16: Crie a função `paddleBounce` que recebe a posição e velocidade da bola, a posição e velocidade horizontais do jogador e retorna a nova velocidade da bola, caso haja colisão. Em caso de colisão, a nova velocidade da bola será $(vx + 0.3 * pv, -vy)$.

```
-- | Detecta colisão da bola com o jogador, alterando sua velocidade.
paddleBounce :: Position -> Position -> Float -> Float -> Position
paddleBounce bp bv pp pv = undefined
```

Exercício 17: Crie a função `blockCollision` que altera a velocidade da bola ao colidir com um dos blocos. Se a bola atingir algum canto horizontal do bloco e sobrepor o bloco verticalmente OU se a bola atingr algum canto vertical do bloco e sobrepor o bloco horizontalmente, a velocidade será alterada para (-vx, vy) ou (vx, -vy), respectivamente. Utilizando a função `hitCornerH` e `hitCornerV`, complete o código abaixo definindo os cantos do bloco com a faixa $(0.8 * bHalfWidth, bHalfWidth)$ na horizontal e $(-bHalfHeight, -0.8 * bHalfHeight)$ na vertical, na definição principal da função utilize `foldl` para mudar a velocidade da bola para cada bloco de `bs`:

```
blockCollision :: Position -> Position -> Blocks -> Position
blockCollision v (xball, yball) bs = undefined
  where
    changeVel (vx, vy) (Block (xb, yb) c) | hitCornerH xb && overlapY yb = (-vx,  vy)
                                          | hitCornerV yb && overlapX xb = ( vx, -vy)
                                          | otherwise                    = ( vx,  vy)

    hitCornerH xb = undefined
    hitCornerV yb = undefined
    overlapY   yb = overlap yballRange $ range yb bHalfHeight
    overlapX   xb = overlap xballRange $ range xb bHalfWidth
    xballRange    = range xball ballSize
    yballRange    = range yball (-ballSize)
```

Exercício 18: Crie a função `removeBlocks` que filtra a lista de blocos mantendo aqueles que não forem atingidos pela bola:

```
-- | Remove blocos atingidos.
removeBlocks :: Blocks -> Position -> Blocks
removeBlocks bs (xball, yball)  = undefined
  where
    hit (Block (xb, yb) c) = overlapBallX (range xb bHalfWidth)
                             && overlapBallY (range yb bHalfHeight)

    xballRange             = range xball ballSize
    yballRange             = range yball ballSize
    overlapBallX           = overlap xballRange
    overlapBallY           = overlap yballRange
```

O Jogo!

Finalmente chegamos a implementação do jogo. Acrescente um arquivo *Game.hs* no projeto e insira o seguinte códigos com as informações do jogo:

```
module Game where

import Graphics.Gloss
import Graphics.Gloss.Interface.Pure.Game
import Window
import Ball
import Blocks
import Player
import Collision

-- Informações do jogo

-- | Estado do jogo
data GameStatus = Game
  { ballLoc   :: Position -- ^ (x, y) coordenada da bola.
  , ballVel   :: Position -- ^ (x, y) velocidade da bola.
  , playerLoc :: Float    -- ^ Posição horizontal do jogador.
  , playerVel :: Float    -- ^ Velocidade do jogador.
  , playerAcc :: Float    -- ^ Aceleração do jogador.
  , isPaused  :: Bool     -- ^ Indicador do status de pausa.
  , blocks    :: Blocks   -- ^ Lista de blocos na tela.
  , gameStat  :: Int      -- ^ Status do jogo: 0 - em jogo, 1 - vitória, -1 - derrota.
  }
```

```haskell
-- | Estado inicial do jogo.
initialState :: GameStatus
initialState = Game
  { ballLoc   = (0, -100)
  , ballVel   = (25, -150)
  , playerLoc = 0
  , playerVel = 0
  , playerAcc = 150
  , isPaused  = True
  , blocks    = map genBlock [0..59]
  , gameStat = 0
  }
```

Exercício 19: Complete as seguintes funções utilizando outras funções implementadas anteriormente:

```haskell
-- | Converte o estado do jogo em uma imagem de tela.
render :: GameStatus -> Picture
render game = pictures [ballPic, walls, playerPic, blocksPic, msgPic]
  where
    ballPic   = undefined
    playerPic = undefined
    blocksPic = undefined
    msgPic    = undefined

-- | Atualiza o estado da bola.
updateBall :: Float -> GameStatus -> GameStatus
updateBall seconds game = game { ballLoc = undefined }
  where pos = ballLoc game
        v   = ballVel game

-- | Atualiza o estado do jogador.
updatePlayer :: Float -> GameStatus -> GameStatus
updatePlayer seconds game = game { playerLoc = undefined }
  where x = playerLoc game
        v = playerVel game

-- | Atualiza posição da bola de acordo com colisões nas bordas.
updateWall :: GameStatus -> GameStatus
updateWall game = game { ballVel = undefined }
  where pos = ballLoc game
        v   = ballVel game

-- | Atualiza posição da bola de acordo com colisões com o jogador.
```

```haskell
updatePaddle :: GameStatus -> GameStatus
updatePaddle game = game { ballVel = undefined }
  where bp = ballLoc   game
        bv = ballVel   game
        pp = playerLoc game
        pv = playerVel game

-- | Atualiza posição da bola de acordo com colisões nos blocos e remove blocos.
updateBlocks :: GameStatus -> GameStatus
updateBlocks game = game { ballVel = ballVel', blocks = blocks' }
  where
    -- atualiza a velocidade da bola ao atingir blocos
    ballVel' = undefined
    blocks'  = undefined
```

Exercício 20: Finalmente podemos criar uma função que atualiza todo o estado do jogo, vamos chamá-la de `update`. Existem quatro condições diferentes que temos que tratar, se o jogo está pausado, retorna o status atual do jogo. Se não existirem mais blocos a serem destruídos, retorna o jogo com `gameStat = 1`, se a bola cair, retorna o jogo com `gameStat = -1`, caso contrário retorna as atualizações de colisão aplicados ao resultado das atualizações de movimento (funções anteriores):

```haskell
-- | Atualiza o estado do jogo.
update :: Float -> GameStatus -> GameStatus
update seconds game | undefined
                    | undefined
                    | undefined
                    | undefined
                    where
                      dropped    = y < (-halfHeight) - 5
                      y          = snd $ ballLoc game
                      collisions = updatePaddle . updateBlocks . updateWall
                      moves      = updatePlayer seconds . updateBall seconds
```

Para capturar o pressionar de teclas criaremos uma função chamada `handleKeys` que recebe um evento, um estado do jogo, e retorna o jogo atualizado. O tipo `Event` é definido como:

```haskell
data Event
    = EventKey    GLUT.Key GLUT.KeyState GLUT.Modifiers (Float, Float)
    | EventMotion (Float, Float)
```

```
    deriving (Eq, Show)
```

representando eventos de teclas e botões ou movimentos do mouse. Para representar que apertamos a tecla da direita, utilizamos o evento `Event (SpecialKey KeyRight) Down 0 0`, por exemplo. Com isso podemos construir a função `handleKeys` utilizando pattern matching:

```
-- | Responde aos e ventos de teclas.
handleKeys :: Event -> GameStatus -> GameStatus
-- Tecla 'r' retorna ao estaod inicial.
handleKeys (EventKey (Char 'r') Down _ _)            game = initialState
-- Tecla 'p' pausa e despausa o jogo.
handleKeys (EventKey (Char 'p') Down _ _)            game = invPause game
-- Tecla '←' move para esquerda.
handleKeys (EventKey (SpecialKey KeyLeft) Down _ _)  game = decVel game
-- Soltar a tecla '←' para o jogador.
handleKeys (EventKey (SpecialKey KeyLeft) Up _ _)    game = incVel game
-- Tecla '→' move o jogador para a direita.
handleKeys (EventKey (SpecialKey KeyRight) Down _ _) game = incVel game
-- Soltar a tecla '→' para o jogador.
handleKeys (EventKey (SpecialKey KeyRight) Up _ _)   game = decVel game
-- Qualquer outra tecla é ignorada.
handleKeys _ game = game
```

Exercício 21: Implemente as funções `invPause`, `decVel` e `incVel`:

```
-- | Incrementa a velocidade do jogador.
incVel :: GameStatus -> GameStatus
incVel game = undefined

-- | Decrementa a velocidade do jogador.
decVel :: GameStatus -> GameStatus
decVel game = undefined

-- | Inverte o estado de pausa do jogo.
invPause :: GameStatus -> GameStatus
invPause game = undefined
```

Finalmente altere o arquivo *Main.hs* para:

```
-- | Arkanoid game implemented in Haskell.
module Main where

import Graphics.Gloss
import Graphics.Gloss.Interface.Pure.Game
import Window
```

```haskell
import Game

-- Propriedades da animação

-- | Número de frames por segundo.
fps :: Int
fps = 60

-- | Window creation.
main :: IO ()
main = play window background fps initialState render handleKeys update
```

Divirtam-se!!